CS 202 - Computer Science II

Project 4

Due date (FIXED): Wednesday, 2/26/2020, 11:59 pm

Objectives: The main objectives of this project is to test your ability to create and use C++ classes, with multiple constructors, static members/functions, and expand to operator overloading. A review of pointers, structs, arrays, iostream, file I/O and C-style strings is also included.

Description:
This project will significantly expand upon Project 3 by adding additional functionality, and implementing more abstract data types (ADTs) and their operations through classes.

- **Pointers must be used for all array manipulation**, including arrays with ADTs (structs, classes) e.g, rental cars, rental agencies. **Pointers must be used in function prototypes and function parameter lists** - not square brackets. Make sure all your C-string functions (e.g. string copy, string compare, etc.) work with pointers (parameters list and function implementation). Square brackets should be used only when declaring an array, or if otherwise you specify your own overloaded `operator[]` . **Pointer arithmetic** (e.g., ++ , +=, - -, -=) and **setting the pointer back to the base address** using the array name **can be used to move through arrays**.
- **Const must be used in parameter lists**, **functions**, and **class method qualification signatures** as appropriate.

The additional functionality is as follows: You are given an updated data file where there is 1 Agency location (**Agency**) which has **5** cars (**Car**) with advanced capabilities. Each car can incorporate **up to 3** (0-3) special driving sensors (**Sensor**). You will have **similar menu options**, but the **functionality has been updated** below. Note: using multiple helper functions to do smaller tasks will make this project significantly easier.

The Sensor Class will contain the following private data members:

- **m_type**, a C-string `char` array of 256 max characters (name of sensor type), valid strings for `Sensor` m_type are `"gps"`, `"camera"`, `"lidar"`, `"radar"`, `"none"`.
- **m_extracost**, a float (additional rent cost per day for the car that carries the sensor, for `"gps"`:=$5.0/day, for `"camera"`:=$10.0/day, for `"lidar"`:=$15.0/day, for `"radar"`:=$20.0/day, for `"none"`:=$0.0/day)
- **gps_cnt**, a `static int` member (keeps track of existing gps-type sensors)
- **camera_cnt**, a `static int` member (keeps track of existing camera-type sensors)
- **lidar_cnt**, a `static int` member (keeps track of existing lidar-type sensors)
- **radar_cnt**, a `static int` member (keeps track of existing radar-type sensors)

and will have the following methods:

- **Default Constructor** – will set the aforementioned data members to default initial values.
- **Parameterized Constructor** – will create a new object based on a C-string value passed into it (sensor type being instantiated). *Hint*: bear in mind what a sensor "type" implies about its other data members.
- **Copy Constructor** – will create a new object which duplicates an input `Sensor` Object.
- **get/set methods** for appropriate data member(s).
- **A get and a reset** `static` member function to return and to reset each of the `static` member variables.

- **A Method to check if 2 Sensor Objects are the same.**
  You must implement this as an operator overload of (`operator==`), and more specifically as Member function of the Class.

The Car Class will contain the following private data members:

- **m_make**, a C-string `char` array of 256 max characters (car make)
- **m_model**, a C-string `char` array of 256 max characters (car model)
- **m_year**, an `int` (year of production)
- **m_sensors**, a `Sensor` class type array of size 3 (max allowable number of sensors per car). *Hint*: You are allowed to use an auxiliary member variable of your choice to keep track of how many actual sensors exist onboard, this will also help for instance in case adding a new sensor is required.
- **m_baseprice**, a `float` (price per day for the sensorless vehicle)
- **m_finalprice**, a `float` (price per day with the increased cost of the car sensors)
- **m_available**, a `bool` (1 = true; 0 = false; try to display true/false using the "`std::boolalpha`" manipulator like: `cout << boolalpha << boolVariable;`)
- **m_owner**, a C- string `char` array of 256 max characters (the current lessee; if no lessee, i.e. the `Car` object is available), set to a `'\0'`-starting (0-length) C-string).

and will have the following methods:

- **Default Constructor** – will set the aforementioned data members to default initial values.
- **Parameterized Constructor** – will create a new object based on the values passed into it for the make, model, year, baseprice, and sensors.
- **Copy Constructor** – will create a new object which duplicates an input `Car` object.
- **get methods** for data members.
- **set methods** for data members except the **m_sensors**, and **m_finalprice**.
- **updatePrice** – a method to update the **m_finalprice** after any potential changes (to the **m_baseprice** or the **m_sensors**)
- **print** – will print out all the car's data.
- **estimateCost** – will estimate the car's cost *given* (a parameter passed to it) a number of days to rent it for.
- **A Method to Add a Sensor** to the `Car` object.
  You must implement this as an operator overload of (`operator+`), and more specifically as Member function of the Class.
- **A Method to Add a lessee (the name of a lessee)** to the `Car` object.
  You must implement this as another operator overload of (`operator+`), again as a Member function of the Class. *Hint*: bear in mind what side-effects on other data members the operation of "adding a renter" to a `Car` object might have.

The Agency Class will contain the following private data members:

- **m_name**, a C-string `char` array of 256 max characters.
- **m_zipcode**, an `int` variable (*Note*: not an array as in the previous Project any more). *Extra Grade Opportunity*: Try to make this a `const int` for +5 extra points. In such a case, the readAllData method (mentioned below) will obviously not be capable of modifying the m_zipcode value.
- **m_inventory**, an array of `Car` objects with a size of 5.

and will have the following methods:

- **Default Constructor** – will set the aforementioned data members to default initial values.
- **get/set (IF possible) methods** for **m_name** and **m_zipcode** data members – **by-Value**.

- **A Method to Index by-Reference an Object of the m_inventory** data (i.e. you should use return by-Reference). *Hint*: This will allow you to access (read and write) to the agency's inventory like in Project_3.)
  You must implement this an operator overload of (`operator[]`). *Reminder*: Any calls to this operator are excluded from the Project's restrictions about using brackets.
- **readAllData** – reads all of the data for the agency from a user-provided file.
- **printActiveSensors** – prints the **TOTAL number of sensors** built to equip the agency's car fleet (total number by sensor type).
- **printData** – prints out to terminal the following data for an agency
  a) name, b) zipcode, c) total number of sensors by sensor type.
- **printAllCars** – prints out all the `Car` Objects of the agency – as well their inventory index (1-5 indexing for the 5 cars).
- **printAvailableCars** – prints out only for the available `Car` Objects of the agency – as well their inventory index (1-5 indexing for the 5 cars).

You should again have a **userMenuPrompt()** function that displays the following **5 possible options** to the terminal, and takes **user input (1-5)** to execute the corresponding functionality. Each option has to be accompanied with a **descriptive prompt** that informs the user what functionality will be executed. Options **outside the possible** range have to be appropriately handled. The User Menu should be **displayed again** after an option is selected and executed (except for the case of option 7 which should terminate the program). The options to implement are:

- **Option 1) Ask** the user for the **input file name**, and then open and **read ALL** data from that **file**. The provided sample file HighTechAgency.txt is **structured** as follows:
  The first line is the car agency info, followed by 5 cars.
  For each car the order is: year make model baseprice {sensors} available [lessee].
  The sensors are enclosed in {braces} and can be 0 up to 3 ws-separated names.
  The lessee name is [optional], it will only be there if the car is available.
  The data have to be stored into **arrays of Class type Objects**.
  You have to declare and implement a function **`readIn()`** that takes whatever parameters are necessary, and completely handles the above described functionality.
- **Option 2) Print out to terminal ALL** data for **the Agency** and **ALL its corresponding Cars** in a way that demonstrates this relationship (see Sample Output section).
  You have to declare and implement a function **`printAll()`** that takes whatever parameters are necessary, and completely handles the above described functionality.
- **Option 3) Print out to terminal ALL** data for **the Agency** and **ONLY its available Cars** in a way that demonstrates this relationship (see Sample Output section).
  You have to declare and implement a function **`printFiltered()`** that takes whatever parameters are necessary, and completely handles the above described functionality.
- **Option 4) Reserve** a Car and **Refresh** list – You should **prompt** for a particular Car (using 1-5 indexing for the 5 cars in the inventory). If possible, you should then update that Car's **lessee and availability status** (obviously you have to prompt for a lessee name as well), and then refresh the list on the terminal by showing all cars. If the car is not available, you should print a warning message to the user.
  You have to declare and implement a function **`reserveOne()`** that takes whatever parameters are necessary, and completely handles the above described functionality.
- **Option 5) Exit** program.

The following minimum functionality and structure is required:

- Ask the **user** for the **input file** name.
- The list of **Sensors** must be stored in an **array of Objects**.
- The list of **Cars** must be stored in an **array of Objects**.
- Use **character arrays** to hold your strings (i.e., C-style) exclusively (using the `string` data type is still not allowed).
- At least one function must use **Pass-by-Reference**.
- At least one function must use **Return-by-Reference**.
- Otherwise, as before, you are free to use **pass by-Value, pass by-Reference**, **pass by-Address** for your function parameters.
- Variables, data members, functions, function signatures, should almost all be made **const** in a perfect design, unless there is no other way for the program to work.
  You must use this guideline to ensure **`const`**-correctness is your class and program design.
- **Pointers** must be used for **all array manipulation** (iterating over elements to read/modify cannot be performed with bracket operator accessing). The only exception on `[]` is if you are using your own overload of `operator[]`.
- **Pointers** must be used in **function prototypes** and **function parameter lists** (the bracket notation is not allowed in parameters lists).
- **Pointers** can only be **moved by incrementing or decrementing** (pointer arithmetic):

```
double d[3] = {1,2,3};
double * d_Pt = d;
for (int i=0; i<3; ++i,++d_Pt){ cout << *d_Ptd; }
```

- Or by **setting** the pointer **back to the base address** using the array name.

```
d_Pt = d;  cout << *d_Pt << endl;
```

- Write your **own C-string length**, **compare**, **copy**, **concatenate** function. Their prototypes will have the form (use the prototypes exactly as provided, with **`char *`** parameters):

```
// counts characters in str array until a NULL-character '\0' is found,
then it returns that number excluding the '\0' one
// the return type size_t represents an unsigned integral number large
enough to contain the maximum possible number of a storage size that
can appear on a target architecture
size_t myStringLength(const char * str);

// returns 0 when the C-strings match, i.e. their characters are equal
one-by-one until a NULL-character '\0' is found in both strings and at
the same position as well
// returns a value <= -1 if the first character that does not match has
a lower value in str1 than in str2
// returns a value >= 1 if the first character that does not match has
a higher value in str1 than in str2
int myStringCompare(const char * str1, const char * str2);

// copies characters from source to destination array until a NULL-
character '\0' is found in source, then it NULL-terminates destination
too
// returns a pointer to the destination array
char * myStringCopy(char * destination, const char * source);

// appends the content of source to the destination array
// this means that the NULL-terminator of destination is overwritten by
the first character of source and a NULL-character '\0' is appended at
the end of the concatenated Cstring in destination
// returns a pointer to the destination array
char * myStringCat(char * destination, const char * source);
```

- Your code has to have the following structure:
  - **proj4.cpp** : The source code file that contains your main() function. It has to include all necessary headers.
  - **my_string.h, my_string.cpp** : The header and source files for your custom library performing Cstring manipulation.
  - **Sensor.h, Sensor.cpp**: The class declaration and implementation files for Sensor objects.
  - **Car.h, Car.cpp**: The class declaration and implementation files for Car objects.
  - **Agency.h, Agency.cpp**: The class declaration and implementation files for Agency objects.
  - **menu.h, menu.cpp** : The header and source files for the functions: **`userMenuPrompt`**, **`readIn`**, **`printAll`**, **`printFiltered`**, **`reserveOne`**.

Important Requirement:

- Your code starting from this project and onwards must follow the **file organization structure** demonstrated in this week's Lab #4.
- Based on this paradigm, you also have to prepare and submit the **Makefile** that handles the build sequence for your entire project.

The following are a list of restrictions:

- Compile your code using either the C++98 or C+03 standard but no higher (**`g++ -std=c++98 ...`** -or- **`g++ -std=c++03 ...`**).
- No usage of external libraries for C-string manipulation is allowed (e.g. **`<cstring>`** **`<string.h>`**), or any **`std::string`** libraries and data types.
- No libraries except **`<iostream>`** and **`<fstream>`** are allowed.
- No global variables except **`const`** ones.
- No usage of dynamic memory.
- You are expected to employ code abstraction and reuse by implementing and using functions. The already provided code structure in the project description will be considered sufficient.
- You are expected to implement **`const`** correctness in your program design. This refers to class method qualifications, function parameter qualifications, etc.

Note: Implement Class Constructors with Member Initializer-Lists (advised)!

This will help later on where you will have to implement constructors this way as certain class data members will required to be const (in this project, it is not a prerequisite, just an opportunity for some extra grade).

Sample Output for menu option 2:

Enterprise 89502

Active sensors: {gps}:1 {camera}:2 {lidar}:3 {radar}:1

[1] 2014 Toyota Tacoma	Base: 115.12	With {gps}: 120.12	Available: true
[2] 2012 Honda CRV	Base: 85.10	With {camera lidar}: 110.10	Available: false
[3] 2011 Toyota Rav4	Base: 65.02		Available: false
[4] 2009 Dodge Neon	Base: 45.25	With {camera lidar radar}: 90.25	Available: true
[5] 2015 Ford Fusion	Base: 90.89	With {lidar}: 105.89	Available: false

Sample Output for menu option 3:

Enterprise 89502

Active sensors: {gps}:1 {camera}:2 {lidar}:3 {radar}:1

[1] 2014 Toyota Tacoma	Base: 115.12	With {gps}: 120.12	Available: true
[4] 2009 Dodge Neon	Base: 45.25	With {camera lidar radar}: 90.25	Available: true

The completed project should have the following properties:

- Written, compiled and tested using Linux.
- It must compile successfully using the g++ compiler on department machines or the provided Xubuntu VM image.
- The code must be commented and indented properly.
  Header comments are required on all files and recommended for the rest of the program.
  Descriptions of functions commented properly.
- A one page (minimum) typed sheet documenting your code. This should include the overall purpose of the program, your design, problems (if any), and any changes you would make given more time.

Turn in: Compressed file structure (with .cpp and .h files, and your Makefile). Also, your project documentation file.

Submission Instructions:

- You will submit your work via WebCampus
- Compress your:
  1. Code file structure (containing Source code files, Header files, Makefile)
  2. Documentation

  Do not include executable or library files (**`bin`** and **`lib`** folders should be there however!)
- Name the compressed folder:
  PA#_Lastname_Firstname.zip
  ([PA] stands for [ProjectAssignment], [#] is the Project number)
  Ex: PA4_Smith_John.zip

Verify: After you upload your .zip file, re-download it from WebCampus. Extract it, compile it and verify that it compiles and runs on the ECC systems.

- Code that does not compile will be heavily penalized –may even cost you your *entire* grade–. Executables that do not work 100% will receive partial grade points.
- It is better to hand in code that compiles and performs partial functionality, rather than broken code. You may use your Documentation file to mention what you could not get to work exactly as you wanted in the given timeframe of the Project.

Late Submission:

A project submission is "late" if any of the submitted files are time-stamped after the due date and time. Projects will be accepted up to 24 hours late, with 20% penalty.